Computer Science Department

TECHNICAL REPORT

A Program Analysis Tool for Evaluating the Ada Compiler Validation Suite

Deborah Rennels
Edmond Schonberg

Technical Report 489

January 1990

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012

A Program Analysis Tool for Evaluating the Ada Compiler Validation Suite

Deborah Rennels
Edmond Schonberg

Technical Report 489

January 1990

A Program Analysis Tool for Evaluating the Ada Compiler Validation Suite *

Deborah Rennels
New York University

Edmond Schonberg
New York University

January 10, 1990

Abstract

The Ada Compiler Validation Capability (ACVC) is a large collection of programs used to verify that compilers conform to the Ada language standard. The maintenance of the ACVC is complicated by ongoing decisions of the Ada Rapporteur Group concerning possible gaps or ambiguities in the language definition. We describe a tool that is intended to simplify the maintenance and upgrade of the ACVC, by providing a specialized data-base front-end to the test suite. This front-end consists of a pattern-matching system, based on the translator of the Ada/Ed system, and index files into the ACVC, built on primary syntactic features of Ada and on previously defined patterns. The patterns used by the system are compact fragments of Ada programs that contain the features of interest. We describe in detail the pattern language, the index files, and the user interface of the system. We expect the system to be particularly useful when the modifications to Ada that will result from the Ada9X project are used to upgrade the ACVC suite.

1 Introduction

The Ada Compiler Validation Capability (ACVC) is used by the Department of Defense to test Ada compilers for adherence to the language standard. The maintenance of this test suite, which currently consists of over 3700 tests (over 190,000 lines of Ada source), in more than 4000 files, is a sizable software engineering project. As ambiguities in the specification (the Ada language definition) are resolved, the test suite must be updated and modified. Such modification involves identifying whether (and where) the language features under consideration occur in the test suite, so that such occurrences can be changed, deleted, or possibly added. The effort involved in such tasks is expected to increase as a result of the Ada9x project, which is currently soliciting suggestions for changes to the language definition [3].

Although the test suite is indexed by section numbers of the Ada Reference Manual [2], identifying where various features occur within the suite is not straightforward. First, the items of interest are often feature combinations rather than solitary features, and as such may be found in many different tests, even ones which are indexed by a manual section not pertaining to these features at all. Second, the features of interest often involve issues of semantic contexts that may not be identifiable from just a textual or syntactic analysis of the test programs. Instead, a full semantic analysis of the program may be required. Finally, the features of interest are not pre-determined, and thus cannot be hard-wired into a test maintenance tool. Successful

*This work was done under subcontract to the Ada Maintenance Organization (AMO) , managed by SofTech, Inc. Contract no. 3451-2-327 maintenance of the test suite depends on being able to identify occurrences of *any* feature combinations that come to be considered important.

This paper describes a program analysis tool, PAT, which has been developed for the ACVC Maintenance Organization (AMO) as an aid in the evaluation and maintenance of the Ada compiler validation suite. The purpose of the tool is to determine which of a set of specified Ada features are present in a given Ada program. The relevant Ada language features are specified by writing an Ada-like program fragment exhibiting these features. This approach allows arbitrary combinations of features to be easily expressed.

2 Compiler Testing and Validation

The ACVC is a large-scale example of specification-based testing. The tests in the ACVC are constructed following a careful but informal analysis (described in the Implementers' Guide [I]) that lists semantic interactions among features of the language, and proposes tests for each of these interactions. (For example: test that lexicographic comparisons on arrays of boolean types work correctly.) In spite of the care with which the IG was drawn, it is still common for Ada users to find unexpected interactions among language features that cause validated compilers to fail. Conversely, each ACVC test contains combinations of features that are not necessarily part of any stated test objective, but that contribute to the general usefulness of the test suite. These combinations of features cannot be described by a context-free grammar, because they invariably involve static and dynamic *semantic* properties of the program such as types and subtypes. Furthermore, there is no way to draw an a priori list of feature interactions: with ≈ 400 primary (syntactic) features in Ada, there are 6.4×10^6 third-order interactions that are potentially interesting. Finally, features interact when they stand in specific syntactic relation to each other: generics interact with a fixed-point-type if, for example, a generic unit with a private type is instantiated with a fixed-point-type. (It is of no interest if the generic unit and the fixed point type appear in different contexts in a program). This indicates that the queries to ACVC tests must be expressed as much as possible in Ada itself. These considerations have led to the following design.

3 Overview of the PAT Tool

The following is a typical query that PAT must be able to answer: "Find all tests that use arrays of fixed-point types". It is easy to see that conventional syntax analysis is insufficient to answer such a query: the fixed type T appearing in the declaration: "**type AF is array(min..max) of T;**" may itself be declared in a separate compilation unit, so that textual, context-free retrieval is unable to recognize instances of the query. As a consequence, PAT operates not on source text, but on the semantically analyzed intermediate representation of the test program. Most useful queries to PAT have the same form: they involve combinations of statement forms and type classes ("... functions returning unconstrained records", "... generic packages with nested instances of generic functions", "... discriminated records containing two dynamic array components", etc.)

The analysis tool is built upon the NYU Ada/Ed interpreter, which is written in SETL, a high-level language based on set theory [4]. Ada/Ed was the first Ada compiler validated by the ACVC (in April 1983) and since then has served informally as an executable model of the Ada language. Because it is written in SETL, and it is an interpreter, the intermediate representation of source programs is fairly easily extracted and modified.

When given both an Ada test program and a PAT program (a "pattern"), the tool determines whether or not the feature combinations specified by the pattern occur within the test program, by the following sequence of steps:

1. Both the PAT pattern and the Ada program are compiled by a modified Ada/Ed into their intermediate code forms. In the case of the legal Ada program, this intermediate form also includes the program symbol table.

2. To answer the basic query as to feature occurrence, a pattern matching routine is invoked on these two intermediate representations. The comparison is done by a set of backtracking tree matching routines, which match the intermediate (tree) forms of the pattern and the test program. These routines access the program's symbol table and thereby allow for matching of semantic features, not simply syntactic ones. Both the patterns and the test programs can of course be pre-compiled, resulting in faster matching at the cost of greater storage requirements for the intermediate forms.

The following is a short example PAT specification to demonstrate the overall approach.

Example 1:

```
procedure main is
    type Rec ( D : pat_integer_type  :=  ...) is record
        ...
        case D is
            ...
        end case;
    end record;
    ...
begin
    ...
end;
```

This pattern specifies a "procedure containing a record type with a variant part, and a discriminant of some integer type with a a default initialization". (The description illustrates the difficulty of specifying the relationships among language features purely textually.)

We chose to keep the PAT input language flexible, requiring no object or type declarations, for example, so that the user (a member of the ACVC team, or a compiler writer preparing for validation) could simply write program "fragments" to specify the features of interest. Whatever information is in the pattern is used as a template for matching against programs, but in most cases not enough information will be given to do a real semantic analysis or type resolution on the patterns. For this reason, we cannot produce a fully annotated syntax tree from the patterns and match these against comparably annotated Ada trees. For example, we cannot distinguish between procedure calls and entry calls in patterns, since neither need be declared before use. Instead, we translate any such 'call' node into a tree form that will match either kind of call in a test program. Other PAT constructs are translated into various "wild-cards": tree forms that either match a number of different Ada subtrees, or that affect the pattern matching process in some particular manner.

The example above demonstrates the three main constructs of the PAT input language (in addition to normal Ada syntax): PAT keywords, pattern variables, and ellipses. Each of these is now briefly described.

a) The word **pat_integer_type** is a PAT *keyword* — it represents the entire class of legal Ada integer types, and will match an occurrence of any such type. This allows the user to specify this semantic feature (integer type definition) without having to overspecify it by giving a particular syntactic representation. Thus INTEGER (the predefined type), a subtype of integer, or any (sub) type derived from INTEGER will match pat_integer_type.

b) Identifiers in the patterns are treated as *pattern variables* during the matching process. That is, once they are matched against a corresponding identifier in the Ada test program being matched, they are then bound to that identifier, and cannot later match a different identifier. This allows the user to specify relationships between individual features in the pattern. For example, the relationships between features in the above pattern are indicated by the identifiers connecting them. This results in the natural semantics one would expect of the patterns (based on usual Ada usage), with a few minor differences.

c) Ellipses ("...") indicate *don't-care* situations, i.e. they specify that this pattern should match a test program even if that program has some additional items, which are thus considered irrelevant with respect to this pattern. In this example, there are ellipses to indicate possible additional declarations, statements, record components, and variant record components. The ellipse is also used here to indicate that the particular value of the discriminant default initialization is irrelevant. These ellipsis are especially important in minimizing the amount of detail that the PAT writer must produce. It is crucial to be able to describe the particular features deemed important, and have the tool find instances of these features within *any* test program. The ellipsis allow us to indicate this arbitrariness of the wider context in a natural manner.

The PAT input language is described in more detail in the next section. Section 6 describes a database system that has been built around PAT. The function of the database system interface, in addition to storing and managing the information extracted by the analysis tool, is to increase the efficiency of tool use by identifying un-matchable program/pattern pairs at a preprocessing stage.

4 The PAT Input Language

4.1 PAT Keywords

In most cases we are interested in finding an instance of a *class* of features, rather than in any specific feature. To specify this in a pattern program we use PAT *keywords*. During the tree matching process, these keywords are not matched literally against test tree items, but instead, each keyword triggers an associated subroutine to determine whether or not the test tree item is an instance of the class denoted by the keyword. For example, the pattern tree item might be a PAT keyword such as PAT_FLOATING_POINT_TYPE: the corresponding subroutine will do a symbol table lookup on the test tree item to see if it corresponds to some floating type. In most cases, the keyword denotes a type class, and the procedure simply checks in the symbol table that the corresponding identifier in the test tree does belong to a type in that class.

These type keywords are used as parts of declarations in PAT patterns, for example, as a type mark. Another group of keywords function as complete declarative items, in order to specify occurrences of representation clauses. The next two subsections describe each of these kinds of keywords. All PAT keywords begin with *"pat_"*, in order to clearly identify them, and avoid the inadvertent use of these keywords for regular identifier names.

4.1.1 Type Keywords

The following group of identifiers are treated as type keywords by the PAT system

- pat_any_type
- pat_numeric_type
- pat_integer_type
- pat_real_type
- pat_floating_point_type
- pat_fixed_point_type
- pat_access_type
- pat_enumeration_type
- pat_scalar_type
- pat_discrete_type
- pat_record_type
- pat_array_type
- pat_constrained_array_type
- pat_unconstrained_array_type
- pat_string_type
- pat_private_type
- pat_task_type

Type keywords can be used in place of subtype indications, type definitions, or variable names. The type keyword PAT_ANY_TYPE is a don't-care keyword, it will match any type.

Subtype Indications: All of the type keywords can be used as subtype indications. During parsing, they are simply identifiers: it is only during the tree matching process that their unique meanings take effect. For example, they can be used as in the following declarations:

Obj : PAT_REAL_TYPE;

or:

subtype *num_type* **is** PAT_NUMERIC_TYPE;

Declarations that include such keywords may involve externally defined types and names as well as those defined locally. For example, in a test program that matched the above PAT fragment, the keyword PAT_REAL_TYPE might match against a type defined in another unit.

Type Definitions: These keywords can also be used as type definitions. This allows us declare a certain *kind* of type, without specifying the details of the type. For example, to declare an unconstrained array type.

we *could* use:

"**type** Arr **is array** (pat_any_type range <>) **of** pat_any_type".

However, even this generalized declaration specifies the number of indices. Instead, we can just use a type keyword:

"**type** Arr **is** pat_unconstrained_array_type".

This will match a declaration of any kind of unconstrained array type.

Variable Names: We can also just use the type keyword itself to constrain the matching in a particular place, rather than declaring a variable or type *name* using the keywords. In such cases, we can use PAT type keywords in place of variable names. For example, the pattern statement

"PAT_INTEGER_TYPE := ..."

is constrained to only match a test program statement in which the left hand side is of some integer type. This is a short-cut to declaring a variable of type PAT_INTEGER_TYPE, and then using this variable name on the left hand side.

4.1.2 Representation Clause Keywords

Another group of keywords can be used to denote representation clauses:

- pat_rep_clause
- pat_length_clause
- pat_size_clause
- pat_storage_clause
- pat_access_storage_clause
- pat_task_storage_clause
- pat_small_clause
- pat_enum_clause
- pat_record_rep_clause

Each of these can be used as a complete declarative item, and as such will match any representation clause in the test program (of the corresponding class.) Another way to specify representation clauses within test programs is to use pattern variables and regular representation clause syntax: as in: "for T'size use ...". Or, to specify a representation clause involving a certain class of type, one can use a type keyword in place of the variable name, as in: "for pat_floating_point_type'size use ...".

4.2 Pattern Variables

Pattern variables allow the binding of several occurrences to the (name of) the same entity. An identifier in the pattern program that is not a keyword is treated as a *pattern variable*. This means that all occurrences of such an item are taken to refer to exactly one item in the test tree. During the pattern matching procedure, the first occurrence of a pattern variable, say **X**, in the features pattern will be bound to the name appearing at the matching position in the test tree. Subsequent occurrences of **X** in the pattern will match successfully only against occurrences of the same name. This allows us to specify a relationship between different fragments of a pattern program, such as compatibility between the declaration and the subsequent use of a certain construct. For example, given the previous pattern declaration:

"**type** Arr **is** pat_unconstrained_array_type",

we can use A in a later declaration of a procedure with an unconstrained array parameter, as in the following pattern:

Example 2:
```
procedure main is
    ...
    type Arr is pat_unconstrained_array_type;
    procedure P ( C : Arr );
begin
    ...
end;
```

Note that this example will only match a program with an explicitly declared unconstrained array type. Alternatively, the type keyword pat_unconstrained_array_type itself could be used as a formal parameter type mark (omitting the type declaration): the parameter would then also match *predefined* types such as STRING.

The following is an example where pattern variables are necessary to specify the feature combination of interest: a declaration of a recursive record type, with a component of type access to that same record type:

Example 3:
```
procedure main is
    type R;
    type ptr is access R;
    type R is record
        ...
        next : ptr;
    end record;
    ...
begin
    ...
end;
```

Note that, in constrast to such pattern variables, when keywords are matched to items in a test program, no binding is made between the keyword and the item. Thus we may have several occurrences of the same keyword in one pattern, each of which may match to a different item in the test program.

Multiple references to the same pattern variable may occur in widely separated parts of the pattern programs. In fact, we can use these pattern variables to express relationships between different compilation units. For instance, one compilation unit may contain a declaration of a certain kind of data item, and another compilation unit might then reference this item.

A special form involving pattern variables is used to specify references to data objects. It is often useful to know that a particular data item is referenced somewhere within a certain region. The declaration of the item may occur in the same region as this reference, or in an outer scope. In PAT, the particular procedure **reference_to** (*some_name*) is used to specify that the data_object *some_name* is mentioned in the corresponding program fragment. When such a statement is encountered during the match procedure, the corresponding test tree is searched for any reference to the name to which *some_name* is bound. If *some_name* has no binding, then the match fails. Such a data reference may occur in either a top-level or a nested statement. The actual argument in the procedure call can be either a simple name or a selected component. The matching works correctly when the selected component is an object in a package named in a with clause, since the bindings from one compilation unit are carried along through matchings of subsequent units.

4.3 The Ellipses

The ellipses construct ("...") can be used to denote "don't-care" items. It can be used in place of a declaration, a statement, or several other syntactic constructs. Most of these constructs are ones that normally occur in lists or sequences, such as basic declarations, statements, record components, etc. Including an ellipse in such a list is useful as otherwise we would have to specify every single item in the list in order to match a test program. Instead, we want to indicate only certain relevant items, and have the matcher ignore any additional items in the test program. A "..." in such a sequence indicates that the list should be viewed as a *sub-list* to be matched against a list in the test program. (More than one ellipse in a sequence has the same meaning as just one.) This is particularly useful within declarative parts and sequences of statements, where we would rarely be looking for an exact item-by-item match between a pattern and program. Thus a typical pattern for a compilation unit is of the form:

```
procedure foo is
    some declarative items
    ...
begin
    some statements
    ...
end;
```

Other uses of the ellipse occur when Ada language feature combinations are being specified. The ellipse can also be used to indicate don't-care items in more narrow contexts than just declarative parts or sequences of statements. The following is a list of all the grammatical constructs which may be replaced by an ellipse.

- component_declaration (LRM 3.7)
- discriminant_specification (3.7.1)
- variant (3.7.3)
- basic_declarative_item (3.9)
- name (4.1)
- primary (4.4)
- case_statement_alternative (5.4)

- simple_statement (6.1)
- entry_declaration (9.5)
- select_alternative (9.7.1)
- exception_handler (11.2)

When an ellipse occurs within a *list* of any of these items, such as a list of record components, it stands for any number of missing items in the list, i.e., the list is in general an "incomplete" list. The list is also taken to be unordered, which is a useful assumption since it is unlikely that users could specify the exact ordering within a list of items that they want to find.

Some of the constructs above occur in contexts where the list may be empty — for example, a declarative section in a subprogram may contain no declarative items at all. In such cases, an ellipse alone is considered to specify "zero or more" such items. For other constructs, such as discriminant specifications or variants, the list cannot be empty, given the surrounding context indicating that such a list occurs at all. For example, given the following record description:

```
type Rec (...) is record
...
end record;
```

a matching test program must have at least one record discriminant, as otherwise the parentheses of the record discriminant should have been omitted. This is also true for case statement alternatives, since, given a case statement, it cannot have an empty list of alternatives. In each case, the context of a single ellipse should make it clear whether it is specifying "zero or more" or "one or more" items.

4.4 Matching within Nested Scopes

As described above, an ellipse in a list of items indicates that this list pattern can match a test list with additional don't-care items. However, all the items specified must occur in the same scope, that which is indicated in the pattern. It is also possible to indicate that one or more of the items in the list can match against items (in the test program) that occur in a nested scope to that indicated in the pattern. This can be specied for a declarative part, with inner scopes such as procedures, packages, etc., or for a sequence of statements, with nested block statements. The matching process will search recursively for the specified items within any such nested scopes.

Such a group of items is specified by enclosing them within curly brackets ("{" and "}"), preceded by a $\gg$. Each item within the curly brackets can match an item in a *different* nested scope. An occurrence of this construct is considered to be followed by an implicit occurrence of an ellipse in the list as well.

This construct allows us to specify and match language features without knowing in advance whether they will occur on the top level of the test program, or within some nested subprogram, package, or block statement. It is particularly useful within statement sequences, since many statements in ACVC tests occur in nested block statements within the main statement section. The following example illustrates the use of the scoping arrow:

```
Example 4:
procedure foo is
     E : pat_enumeration_type;
     int : pat_integer_type;
     ...
begin
     ≫
     {    E := ...;
          int := ...;
     }
end;
```

This specifies a procedure with a declaration of two variables: one of an enumeration type, the other an integer type. And somewhere in the statement section, an assignment to each of these variables. However, the arrow in the statement section of the procedure indicates that these assignment statements may occur nested anywhere within the statement section, perhaps in a block statement. Without this arrow, the assignment statement would have to occur on the top level of the statements section. This pattern will match either of the following two Ada programs (among others):

```
Example 5:
procedure foo is
     type A_Type is array(1..10) of float;
     Arr : A_Type;
     type small_enum is (a, b, c, d, e);
     small_var : small_enum;
     I : integer;
begin
     Arr(3) := 1.0;
     small_var := a;
     I := 1;
end;
```

```
Example 6:
procedure foo is
     type A_Type is array(1..10) of float;
     Arr : A_Type;
     type small_enum is (a, b, c, d, e);
     small_var : small_enum;
     I : integer;
begin
     begin
          begin
               small_var := a;
          end;
          I := 1;
     end;
end;
```

In the first program, both of the assignment statements occur in the main sequence of statements in the procedure. In the second program, the assignment statements each occur in two different nested scopes within the sequence of statements. This illustrates that the items following the arrow can each match test items found in different inner scopes.

4.4.1 Arrows with Compilation Units

The above use of the arrow construct allows for the specification of undetermined scoping *within* a unit, but the outermost scope is still clearly specified. However, in many cases we may not care at all about the outer context within which a particular construct is located. In such cases, we can simply specify a library unit preceded by a scoping arrow (≫). This indicates that this library unit may be matched in the test program by a program unit found at *any* scope level, either at the top level, or nested. For convenience, in addition to library units, we also allow this for any of the following: *task_declaration*, *task_body*, *package_body*, or *body_stub*.

As an example, this pattern will match any corresponding Ada procedure, nested at any level within a program:

Example 7:

```
≫
procedure foo is
    A : pat_integer_type;
    B : pat_integer_type;
    ...
begin
    A := ...;
    ...
end ;
```

4.5 The PAT Block Construct

The above new constructs allow for the designation of complete library units (and a few other declarative items) without specifying the outer scope. But we also may want to specify just a list of declarations, or a list of statements, without specifying even what kind of unit they may occur in. For this reason, we have added a new production to the Ada grammar, a *pat_block*. Syntactically, this construct is similar to a block statement preceded by an arrow:

```
≫
PAT_BLOCK
    declarative_part
BEGIN
    sequence_of_statements
    [exception
        exception_handler
        {exception_handler}]
END ;
```

The semantics of this construct are such that we can specify a list of declarations, or statements, independently of any surrounding context. Or, we can specify a pairing of declarations and statements, as within a regular block statement. Specifically, the rules are:

1. If the statements section consists of only a "...", then we search only for items occurring in the declarations section.

2. If the declarations section consists of only a "..." (or is entirely empty), then we search only for items occurring in the statements section.

3. If both sections contain anything other than "...", then we search for a scope with this combination of declarations and statements, for example, in a procedure, package body, or block statement.

When there are non-ellipse items in either the declarations or statements sections, we assume an implicit "..." in each section as well.

5 The Ada Features Identification System

The Program Analysis tool has been developed as part of an Ada Features Identification System (AFIS) that also includes facilities for identifying, recording, and retrieving information about the individual language features found in each ACVC test. A set of 295 *primary features* has been defined: a modified Ada/Ed front-end reports which of these features are present in any given Ada program or PAT pattern. A database system is under development to manage the storage and retrieval of this data.

The main purpose of identifying primary language features is to pare down, or "filter" the set of candidate Ada programs that might match a given pattern, and thus make most efficient use of the PAT pattern matching tool. For example, the pattern in example 1 contains the following primary features:

- component_declaration
- default_expression:discriminant
- discriminant_specification
- name
- procedure_specification
- record_type_definition
- subprogram_body
- type_declaration:full
- variant_part

We would not attempt to match this pattern against any Ada programs that do not contain at least all of these features, for example, a program that has no discriminant specification. By first querying the primary features database we can avoid attempting such futile matches.

Note that the primary features occurring in a program or pattern are identified as a *set* of features, in no particular order or context within the unit. Referring back to the example given at the end of section 2, the database may contain the information that a given program contains a generic unit as well as a fixed point type; the full PAT tool must be used to determine whether these two occur in the specific syntactic relation of interest.

The list of primary features was derived primarily from terms used in the Ada LRM, particularly the index and syntax summary (appendix E) of the LRM. In most cases, the features are either nonterminals of the syntax summary, or major terms of the index (those printed in boldface), or both. In some cases, one feature is a general or basic term, and a few other features are special cases of that general feature. For example, the feature "generic_formal_type" has subcases "generic_formal_type:discrete_type", "generic_formal_type:integer_type", etc. Essentially, we have taken some very simple *combinations* of features and considered them as atomic primary features, for the purpose of improving the efficiency of the filtering process. Clearly, the distinction between primary features and feature combinations is a matter of definition. However, the goal of the primary features identification system was to allow identification of features as a simple by-product of (instrumented) parsing and static semantic analysis, so that a database of feature occurrences could be built for the entire ACVC suite by a process no more difficult than a validation run. We have therefore included primary features that are detailed enough to provide effective filtering, but that are easily identifiable from local, mostly syntactic information. More complicated combinations of features require the use of the PAT tool.

The attached figure illustrates the various components of AFIS: the Features Coverage Facility, the database of primary features (PRIM) and its interface, the PAT tool itself, and the database MAPS, which holds information as to previous pattern/ACVC tests matches. The arrows in the figure indicate the actions taken when a new feature combination needs to matched against the ACVC suite. First, the PRIM database is queried for potentially matching tests, based on the primary features extracted from the pattern. The resulting subset of ACVC tests is then matched against the pattern, using PAT. If any matches are found, that test_name/pattern_name pair is stored in MAPS.

In addition to its use as a filter, the primary features information also provides a much more fine-grained indexing into the suite than that afforded by the ACVC naming conventions. The primary features database interface can easily answer such queries as: "list all the ACVC tests that contain generic formal objects with default expressions", or "allocators with qualified expressions".

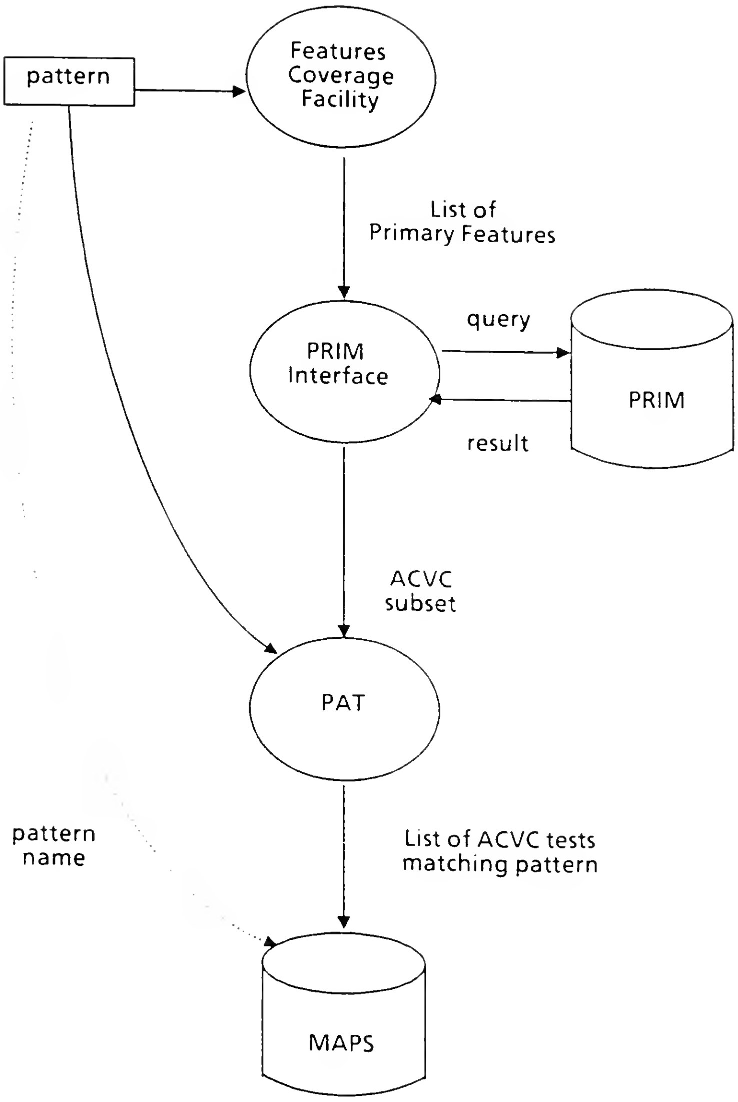
References

[1] United States Department of Defense. *Implementers' Guide — Draft, Version G1*. United States Department of Defense, 1982.

[2] United States Department of Defense. *Reference Manual for the Ada Programming Language*. Springer-Verlag, 1983.

[3] Department of Defense Language Control Facility. Ada language revision initiated. *Language Control Facility Ada-Jovial Newsletter*, 10(4), 1988.

[4] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming With Sets: An Introduction to SETL*. Springer-Verlag, 1986.


pattern
Features Coverage Facility
List of Primary Features
PRIM Interface
query
result
PRIM
ACVC subset
PAT
pattern name
List of ACVC tests matching pattern
MAPS

NYU COMPSCI TR-489
Rennels, Deborah
A program analysis tool for
evaluating the Ada
compiler validation...c.2

DATE DUE	BORROWER'S NAME

DATE DUE
